QoS Negotiation in Real-Time Systems and Its Application to Automated Flight Control

Tarek F. Abdelzaher, *Member*, *IEEE*, Ella M. Atkins, *Member*, *IEEE*, and Kang G. Shin, *Fellow*, *IEEE*

Abstract—Real-time middleware services must guarantee predictable performance under specified load and failure conditions, and ensure graceful degradation when these conditions are violated. Guaranteed predictable performance typically entails reservation of resources and use of admission control. Graceful degradation, on the other hand, requires dynamic reallocation of resources to maximize the application-perceived system utility while coping with unanticipated overload and failures. We propose a model for quality-of-service (QoS) negotiation in building real-time services to meet both of the above requirements. QoS negotiation is shown to 1) outperform "binary" admission control schemes (either guaranteeing the required QoS or rejecting the service request), 2) achieve higher application-perceived system utility, and 3) deal with violations of the load and failure hypotheses. We incorporated the proposed QoS-negotiation model into an example real-time middleware service, called **RTPOOL**, which manages a distributed pool of shared computing resources (processors) to guarantee timeliness QoS for real-time applications. In order to guarantee timeliness QoS, the resource pool is encapsulated with its own schedulability analysis, admission control, and load-sharing support. This support differs from others in that it adheres to the proposed QoS-negotiation model. The efficacy and power of QoS negotiation are demonstrated for an automated flight control system implemented on a network of PCs running **RTPOOL**. This system is used to fly an F-16 fighter aircraft modeled using the Aerial Combat (ACM) F-16 Flight Simulator. Experimental results indicate that QoS negotiation, while maintaining real-time guarantees, enables graceful QoS degradation under conditions in which traditional schedulability analysis and admission control schemes fail.

Index Terms—Quality-of-service (QoS), QoS negotiation, QoS levels and rewards, schedulability analysis and admission control, automated flight systems.

1 Introduction

Predictability in real-time applications is often achieved by reserving resources and employing admission control under a priori assumed load and failure conditions. Graceful QoS degradation, on the other hand, requires dynamic resource reallocation in order to cope with changing load and failure conditions while maximizing system utility. Both predictability and graceful QoS degradation are necessary for real-time applications, but pose conflicting requirements.

The main focus of this paper is on how to achieve predictability and graceful degradation in *long-lived* real-time services for embedded applications. By "long-lived" we mean that a request, if granted, will hold its reserved resources for a relatively long period of time. To control the load imposed on system resources and, hence, guarantee a certain level of QoS, the request must go through admission control and resource reservation. Conventional admission control schemes make "binary" decisions on whether to guarantee or reject each request. Future requests may be rejected because resources have already been committed to those that arrived earlier. In hard-real-time systems, a static analysis may be performed to guarantee a priori that all requests be honored under the assumption of the worst-case request arrival behavior and service requirements. If these assumptions are violated at run-time due to transient overload or resource loss (failures), the guarantees may become invalid, which may, in turn, lead to system failure.

We propose a mechanism for QoS (re)negotiation as a way to ensure graceful degradation in cases of overload, failures, or violation of pre-run-time assumptions. This mechanism permits clients to express in their service requests a *spectrum* of QoS levels they can accept from the provider and perceived utility of receiving service at each of these levels. As a result, the application designer will be able to express acceptable compromises in QoS and their relative cost/benefit as derived from application domain knowledge.

We incorporate the proposed QoS negotiation into a processing capacity management middleware service called RTPOOL. The service is designed and implemented to support timeliness guarantees for a flight control application in which a set of flight control tasks, their QoS levels, and the corresponding rewards are provided by the flight *mission planner* and can be renegotiated, if necessary, using RTPOOL's QoS-negotiation support. The mission planner was developed in the context of the Cooperative Intelligent Real-time Control Architecture (CIRCA) ([1], [2]), which computes task execution trade-offs from application

- *T.F. Abdelzaher is with the Department of Computer Science, University of Virginia, PO Box 400470, Thornton Hall, Charlottesville, VA 22904-4740. E-mail: zaher@cs.virginia.edu.*
- *E.M. Atkins is with the Aerospace Engineering Department, University of Maryland, 3182 Glenn L. Martin Hall, College Park, MD 20742. E-mail: atkins@eng.umd.edu.*
- *K.G. Shin is with the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122. E-mail: kgshin@eecs.umich.edu.*

Manuscript received 11 Aug. 1997; accepted 21 Feb. 2000.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 112756.

domain knowledge and alters the mission plan as required during QoS negotiation.

In this paper, we begin with a review of related work (Section 2), followed by a description of the proposed QoS-negotiation model (Section 3). Next (Section 4), we describe RTPOOL, a distributed processing resource management service that follows the proposed QoS-negotiation model, highlighting the synergy between RTPOOL components and QoS-negotiation support. We present details of RTPOOL implementation and negotiation API (Section 5), then describe the use of RTPOOL in the context of automated flight control (Section 6). Flight performance is evaluated (Section 7), illustrating the efficacy of QoS-negotiation support, followed by a brief paper summary (Section 8).

2 Related Work

Predictable performance of real-time services has traditionally been achieved using resource reservation and admission control. In hard real-time systems, sufficient resources are reserved a priori for the application. Off-line schedulability analysis is used to verify that the reserved resources are sufficient for meeting all timing constraints. Such an analysis requires that the worst-case load/failure conditions be known at design time. For example, the authors of [3] described an optimal schedulability analysis algorithm for uniprocessors, which considers precedence and resource constraints. In [4] and [5], a similar optimal result is derived for multiprocessors, while, in [6], the result is extended to distributed systems. Pre-run-time resource allocation algorithms have been reported for embedded applications such as process control [7], [8], turbo engine control [9], autonomous robotic systems [10], and avionics [11]. AI-based approaches that utilize application domain knowledge are described in [7], [10], [11]. Solutions to the offline schedulability analysis problem have been presented for specific hardware topologies such as hypercubes [12], hexagonal architectures [13], and mesh-connected systems [14]. Simulated annealing [15] has been proposed as an optimization heuristic. Different flavors of using simulated annealing in the context of real-time task assignment and scheduling can be found in [16], [17], [18], [19]. In [20], [21], [22], efficient methods are considered for offline allocation of periodic tasks to computing resources where different tasks may have different deadlines. The above algorithms are static in nature in that they require an exact *pre-run-time* characterization of worst-case offered load and processing capacity. For some applications, the worst-case conditions may be difficult to predict accurately at design time. This is true, for example, of military applications, where it is difficult to characterize and bound a priori the extent of damage on the computing system at run-time. A mechanism is therefore needed to ensure predictable graceful degradation of system performance when the design-time load or failure hypotheses are violated.

Predictability in dynamic real-time systems where load patterns are not known in advance has often been achieved via on-line admission control. Communication services with end-to-end QoS guarantees are one example where on-line admission control is used [23], [24]. Graceful degradation has often been addressed in the context of communication architectures to support QoS maintenance and negotiation for multimedia applications. Examples include the QoS-A framework [25], the Heidelberg QoS model [26], COMETS's Extended Integrated Reference Model (XRM) [27], the OMEGA end-point architecture [28], and the QoS Broker [29]. A good survey of these and other communication architectures is found in [30]. Our work is complementary to these efforts in the sense that we consider a QoS negotiation model suitable for embedded systems and not focused on multimedia applications. While multimedia applications are dominated by high volumes of communicated data whose source and destination are typically fixed, in embedded systems (e.g., process control), computation is more dominant and dynamic task allocation for better load sharing is an important concern.

Predictability in dynamic real-time systems has been addressed outside the communication subsystem as well. The concept of on-line admission control has been applied to resource reservation for dynamically arriving real-time tasks. Many such efforts appear in the context of real-time operating system research. Temporal isolation of real-time applications has been proposed via resource reservation [31], [32], [33], proportional-share resource management [34], and hierarchical CPU scheduling [35]. For hard real-time tasks, the Spring Kernel [36] innovated a new form of plan-based scheduling and on-line admission control guarantees. The Dreams real-time system [37] extends the notion of on-line guarantees further to accommodate transient periodic processes which arrive dynamically and request periodic service throughout a given interval of time. The Rialto operating system [38], which targets multimedia applications, takes the approach of dynamically maximizing aggregate system "value." Clients request their required resources from a resource planner whose goal is to compute a resource allocation that maximizes the user's perceived utility of the system. The Nemesis operating system designed in the context of the Pegasus project [39] investigates support for adaptive multimedia applications. Other real-time operating systems, such as Alpha [40] and Mach [41], export a simple priority-based or value-based interface to allow best effort maximization of overall perceived utility of the system by serving the "most important" tasks first. A suitable run-time scheduling policy [42], [43] can then be used to maximize the total achieved utility/reward. Our work is different in that it does not require changes to the operating system. We consider the design of QoS adaptive middleware services on top of best effort operating system support for embedded applications, rather than investigating operating system design for QoS adaptation. We believe that our approach makes an implementation of our architecture more portable, albeit potentially less efficient.

Compromises between resource reservation for irrevocable service guarantees and best effort maximization of the overall system utility have been addressed. Virtual clock-based communication schemes [44], for example, delay reserving resources for packet transmission until a virtual arrival time, which results in increasing overall system utility over simple FIFO transmission by enforcing a global

priority order. A similar approach is applicable to dynamic real-time tasks. To prevent rejecting important incoming tasks because of lower priority ones holding necessary resources, resource reservation for incoming tasks is *delayed* to a "virtual" arrival time. The delay allows for "more important" tasks to arrive and be served first. Unfortunately, this delay in making task guarantees may itself waste processing bandwidth which may reduce schedulability and increase the rate of task rejections. Instead, we use service QoS as the dimension to trade. QoS negotiation extends the typical real-time service interface in two different ways. First, it offers QoS degradation as an alternative to denial of service, thus enhancing the percentage of accepted service requests and the total perceived system utility. Second, it provides a generic means of utilizing application-specific knowledge to control QoS degradation.

Predictable graceful degradation has also been addressed in the context of fault-tolerant real-time computing. For example, the imprecise computation technique [45] prevents timing faults and achieves graceful degradation by making sure that an approximate result of an acceptable quality is available by the deadline if the exact result cannot be obtained. The tolerance of real-time applications to QoS violations has been exploited in several research efforts. For example, in [46], an overload management technique is discussed for real-time control applications that discards selected task instances upon failures while maintaining satisfactory control loop performance. An adaptable use of redundancy in safety-critical applications is described in [47] to optimize resource utilization and allow graceful degradation of the system in case of failures. A scheduling algorithm that satisfies timing and dependability constraints of mandatory tasks while maximizing system utility by proper scheduling of optional tasks is described in [48]. Our scheme is more general in that we do not investigate a particular application-dependent degradation policy. Instead, our QoS negotiation API allows defining QoS parameters of arbitrary semantics, specifying how these parameters may be degraded, and quantifying the effect of degradation on system utility. We provide a generic framework for achieving graceful degradation of embedded real-time middleware, and describe an application of the generic QoS-negotiation framework to automated flight control for illustration.

3 QoS-Negotiation Model

A simple yet expressive QoS-negotiation model is the key to building predictable, gracefully degradable middleware services for real-time applications. In this section, we describe the application model, the proposed QoS-negotiation model, and the model of a real-time middleware service that supports QoS negotiation. We consider a class of embedded real-time systems in which various software components perform tasks to accomplish a single overall "mission." We will henceforth call this mission an *application*. Flight control, shipboard computing, automated manufacturing, and process control applications generally fall under this category. The application is composed of a set of tasks, each of which requires a set of resources/services. We are concerned mainly with long-lived services that need to hold reserved resources for an extended period of time, such as processor capacity reservation [49] and communication connection establishment services [24].

Our negotiation model is centered around three simple abstractions: *QoS levels*, *rewards*, and *rejection penalty*. A client requesting service specifies in its request a set of *negotiation options* to the service provider and the penalty of rejecting the request, derived from the expected utility of the requested service. Each negotiation option consists of an acceptable QoS level for the client to receive from the provider and a reward value commensurate with this QoS level. The QoS levels are expressed in terms of parameters whose semantics need be known only to the client and the service provider. For example, in establishing a real-time communication connection, these parameters may specify the client's traffic delay and jitter requirements. In processor capacity reservation, they may express the required processor bandwidth, while, in a multicast protocol, they may represent the semantics of the requested multicast service, such as reliable, ordered, causal, or atomic delivery. The reward represents the "degree of satisfaction" to be achieved from the QoS level (i.e., the application-perceived utility of supplying the client with that level of service). Thus, the client's negotiation options represent a set of alternatives for "acceptable" QoS and their "utility". The *rejection penalty* of a client's request is the penalty incurred to the application if the request is rejected. Rejection penalty plays no further role if the request is guaranteed. In Section 6, we describe how QoS levels, negotiation options, and rejection penalty are computed in the context of a flight control application using a mission planner. The planner computes QoS levels, rewards, and penalties from application domain knowledge and a specification of system failure probabilities.

To control system load in a way that ensures predictable service, the service provider must subject the client's request to on-line admission control, which determines whether to guarantee or reject the request. We propose a slightly different notion of guaranteeing a request, as compared to the conventional notion of guarantee. In our model, *guaranteeing* a client's request is the certification of the request to receive service at *one* of the QoS levels listed in its negotiation options. The selection of the QoS level it will actually receive, however, is up to the service provider. Furthermore, the service provider is free to switch this QoS level to another level in the client's negotiation options if it increases perceived utility. Note that specifying only one negotiation option with default (e.g., infinite) rejection penalty reduces this mechanism to traditional on-line guarantee schemes. Thus, while the proposed mechanism should perform no worse than these schemes in the special case, it provides the means to express and take advantage of more accurate semantic information about the application whenever such information is available. In other words, while we do not *require* the application designer to supply more information than is necessary for traditional on-line guarantee schemes, we *offer the flexibility* to take advantage of additional semantic information when it is available. In Section 6, we give an example application that benefits from

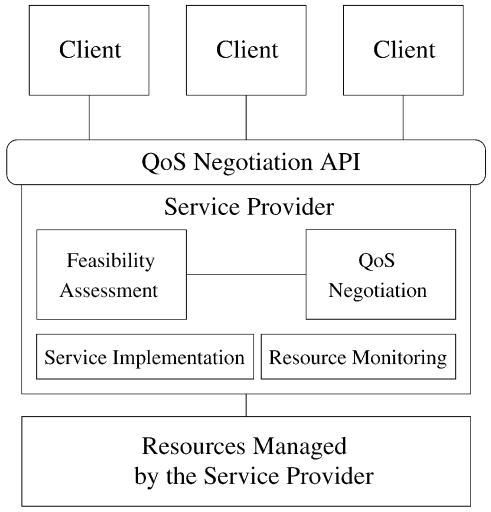

Fig. 1. Service provider architecture.

our support. Shifting the authority in selecting clients' QoS levels from the client to the service provider has two important advantages.

- The application code is decoupled from the assumptions on underlying resource availability and capacity. Such assumptions are implied when a client asks specifically for a certain QoS level. Instead, the client supplies a set of QoS options, along with their application-perceived utility. The service provider then determines QoS levels that are feasible with the resources available and selects the ones that optimize the overall application-perceived utility. Note that this optimization must consider all current clients of the provider (and potentially adjust their QoS levels in anticipation of new requests that may arrive later). Thus, only the provider has the global information required for this optimization. Decoupling application code from assumptions about the underlying resource capacity and letting the service provider optimize system utility subject to resource constraints makes the application more adaptable to variations in resource capacity/availability. Graceful degradation comes naturally out of this property.
- Incoming requests are guaranteed in the order of their arrival (i.e., FIFO), so resources are committed to clients in FIFO order. However, requests from high-priority clients to a service provider should be able to force less important clients holding the necessary resources to degrade their QoS, if possible. Providing negotiation options and delegating QoS level selection to the provider gives the flexibility to adjust QoS levels, when necessary, thereby achieving higher overall system utility while maintaining each client's QoS guarantee at one of the levels specified in the negotiation options.

The QoS-negotiation architecture of the service provider is given in Fig. 1. The provider runs on top of a pool of resources whose size may vary dynamically and serves a dynamic set of real-time clients. The underlying resources available to the provider are monitored by the resource monitoring module. The provider exports a QoS-negotiation API to its clients based on QoS levels, rewards and penalties. The QoS-negotiation module is responsible for selecting the appropriate QoS level for each client so that overall utility is maximized. The feasibility assessment module is responsible for checking whether or not the selected QoS levels of the respective clients can be sustained using currently available resources. Assisted by the feasibility assessment module, the QoS-negotiation module performs admission control on incoming service requests.

4 RTPOOL—Realizing QoS Negotiation

We designed and implemented an example middleware service, **RTPOOL**, to support the proposed QoS-negotiation model. This service is responsible for managing a distributed pool of computing resources (processors) to guarantee timeliness, as illustrated in Fig. 2. It employs a *processor membership protocol* to keep track of processor pool

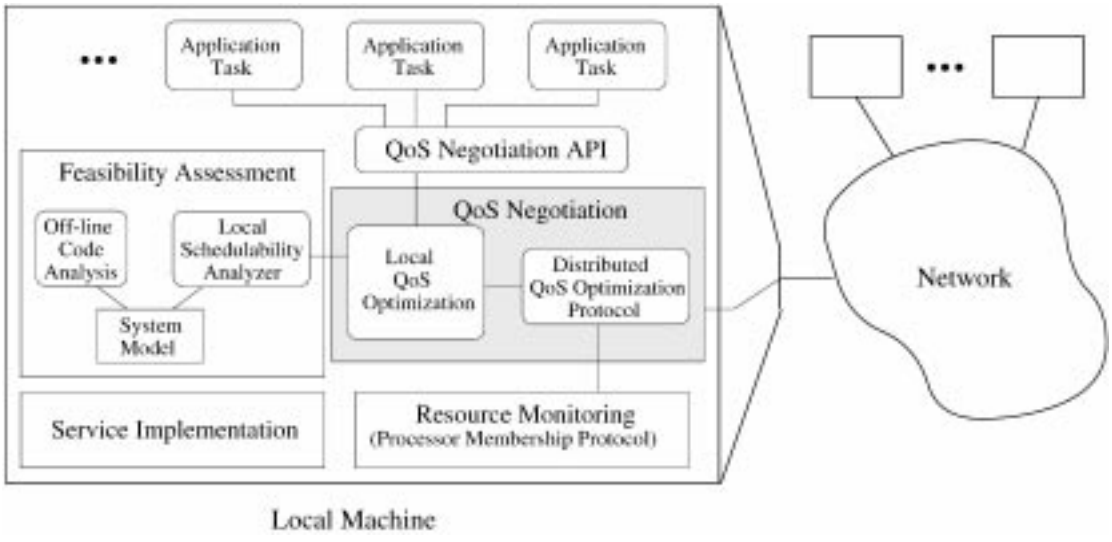

Fig. 2. General overview of **RTPOOL**.

Let each client task T_i have QoS levels $M_i[0], \ldots, M_i[best_i]$ with rewards $R_i[0], \ldots, R_i[best_i]$, respectively.

1. Start by selecting the best QoS level, $M_i[best_i]$, for each client T_i.
2. While the set of selected QoS levels is *not* schedulable, do Steps 3 and 4.
3. For each client T_i receiving service at level $M_i[j] > M_i[0]$, determine the decrease of local reward, $R_i[j] - R_i[j-1]$, resulting from degrading this client to the next lower level.
4. Find client T_k whose $R_k[j] - R_k[j-1]$ is minimum and degrade it to the next lower level.
5. Go to Step 2.

Fig. 3. Local QoS optimization heuristic.

membership and report processor failures. Schedulability analysis is used to provide timliness guarantees. We assume that, although task arrival patterns are not known a priori, application code of an embedded system is available before the system is deployed. Thus, task computing requirements may be characterized off-line (e.g., using profiling tools or compile-time support). Additionally, we integrated support for QoS negotiation into **RTPOOL**. This support is split into local and distributed algorithms and is the focus of this section.

Clients of **RTPOOL** are application tasks. **RTPOOL** service requests are used to guarantee the timeliness of new incoming tasks. Our task execution model is influenced by the requirements of the flight control application (see Section 6), but it is still sufficiently general for use in other applications. **RTPOOL** assumes periodic tasks and handles aperiodic tasks as periodic servers. A task is composed of a set of modules and has a deadline by which all of its modules must be completed. The modules may have arbitrary precedence constraints among themselves specifying their execution sequence. We assume that task arrivals are independent, so we do not support precedence constraints among *different* tasks.

Each request for guaranteeing a task includes its rejection penalty and the negotiation options of the client task that specify different QoS levels and their respective rewards. A client task's QoS level is specified by the parameters of its execution model. For an independent periodic task, the parameters consist of task period, deadline, and execution time. We model period and deadline as negotiable parameters. This represents a significant departure from most scheduling literature, although the authors of [50] articulate on the alterability of task periods in real-time control systems using system stability and performance index. Task execution time, on the other hand, depends on the underlying machine speed and thus should not be hardcoded into the client's request. Instead, each QoS level in the negotiation options specifies which modules of the client task are to be executed at that level. This allows the programmer to define different versions of the task to be executed at different QoS levels or to compose tasks with mandatory and optional modules. The reward associated with each QoS level tells **RTPOOL** the utility of executing the specified modules of the task with the given period and deadline. In Section 6, we present the task set of our application, along with the negotiation options of each task as an example of using **RTPOOL**'s support for QoS negotiation.

Requests for guaranteeing tasks may arrive dynamically at any machine in the pool. Since, in the proposed QoS-negotiation scheme, tasks normally receive higher QoS than their minimum functionality QoS level, it is highly probable for the new arrival to be guaranteed at the local machine. To guarantee a request at the local machine, **RTPOOL** executes a *local QoS-optimization heuristic*. The heuristic (re)computes the set of QoS levels for all local clients (including the new one just arrived) which maximizes the sum of their rewards. Recomputing the QoS levels may involve degrading some tasks to accommodate the new one. The task is rejected if *both* 1) the new sum of rewards (including that of the newly arrived task) is less than the existing sum prior to its arrival *and* 2) the difference between the current and previous sums is larger than the new task's rejection penalty. Otherwise, the requested task is guaranteed. As a result, task execution requests will be guaranteed unless the penalty from resulting QoS degradation of other local clients is larger than that from rejecting the request. When a task execution request is rejected by the local machine, one may attempt to transfer and guarantee it on a different machine using a load-sharing algorithm. Note that conventional admission control schemes (which do not support negotiated QoS degradation) would always incur the request rejection penalty whenever an arrived task makes the set of current tasks unschedulable. By offering QoS degradation as an alternative to rejection and by using admission control rules, we can show that the reward sum (or perceived utility) achieved using our scheme is *lower bounded* by that achieved using conventional admission control schemes given the same schedulability analysis and load sharing algorithms. Thus, in general, our proposed scheme achieves higher perceived utility.

Fig. 3 gives an example of the local QoS-optimization heuristic. The heuristic implements a gradient descent algorithm, terminating when it finds a set of QoS levels that keeps all tasks schedulable, or when it finds the task set unschedulable even at the lowest QoS level of each task, in which case the request is rejected. This heuristic degrades the tasks' QoS in a way to locally minimize the resulting decrease in local reward. Note that, unless all tasks are executed at their highest QoS level, the machine suffers from *unfulfilled potential reward*. The unfulfilled potential reward, UPR_j, on machine N_j, is the difference between the total reward achieved by the current QoS levels selected on

1. On the source machine, N_i, find a client T_k whose removal will result in the maximum increase, W, in the rewards of the remaining clients due to improvement in their QoS level.
2. N_i communicates a service re-assignment request of client T_k, with reward W, to every other machine N_j that satisfies the inequality $UPR_i - UPR_j > V$.
3. Every machine N_j which receives the request, considers T_k as a new arrival and tentatively runs the local QoS optimization heuristic discussed earlier to find new QoS levels for local clients and T_k. If the total reward of the new QoS assignment is higher than the one currently achieved, N_j accepts client T_k, and replies to N_i with the increase W_j of N_j's local reward resulting from its acceptance.
4. N_i transfers T_k to the machine which replied with the maximum W_j.

It is easy to prove that W_j represents the exact increase in global reward resulting from the task transfer.

Fig. 4. Distributed QoS optimization protocol.

the machine and the maximum possible reward that would be achieved if all local tasks were executed at their highest QoS level. This difference can be thought of as a fractional loss to the mission. Often, this loss is unavoidable because of resource limitations. However, such loss may also be caused by poor load distribution, in which case it can be improved by proper load sharing.

RTPOOL employs a load-sharing algorithm that implements a distributed *QoS-optimization protocol*. The protocol uses a hill climbing approach to maximize the global sum of rewards across all clients in the distributed pool. It is activated between two machines N_i and N_j when the difference $UPR_i - UPR_j$ exceeds a certain threshold V. The protocol is given in Fig. 4.

Close examination of the local QoS optimization heuristic and the distributed QoS optimization protocol reveals that neither makes assumptions about the nature of the client and the semantics of its QoS levels.[1] For **RTPOOL** this means complete independence between the task model used by the feasibility assessment module and the QoS-negotiation mechanism. As a result, it is easier to enhance **RTPOOL** to handle more elaborate task models, constraints, and QoS-level parameters/semantics without affecting its QoS-negotiation mechanism. The disadvantage of this separation of concerns compromises optimality somewhat, as illustrated by example in Section 7.

5 Implementation and API

In this section, we highlight implementation details of the **RTPOOL** service, particularly those related to its QoS-negotiation API. **RTPOOL** is currently running on a PC platform using the MK7.2 microkernel from the Open Group[2] The microkernel is a derivative of CMU RT-Mach. **RTPOOL** is implemented as a user-level library which exports the abstraction of tasks, threads, QoS levels, and rewards. Highlighted below are the components of the implemented prototype.

1. The distributed QoS-negotiation protocol, however, assumes service to a given client can be migrated to another node.

2. Open Group was previously known as the Open Software Foundation.

5.1 Support for Scheduling and QoS Negotiation

Our scheduling and QoS negotiation support is implemented as a thread package called *qthreads*. The OG MK7.2 microkernel provides support for creating thread pools that can be time-shared, scheduled FIFO, or scheduled round-robin. Threads can be assigned fixed priorities within a given range. In order to use other scheduling policies, such as deadline monotonic or EDF, we implemented a user-level *local scheduler* that runs on each machine on top of kernel threads. The local scheduler supports *periodic thread* creation with a period that can be changed at run-time in response to changes in the QoS level.

The *qthreads* package is novel in that it exports the abstraction of tasks with associated QoS levels and rewards. Its API permits the user to create tasks, create threads within each task, define QoS levels for the task, and specify rewards. It also permits the user to specify, for a given thread, the QoS levels in which the thread is eligible to execute. The package exports a *force_negotiation()* primitive to initiate QoS negotiation. When new load (i.e., task or a set of tasks) arrives and is to be admitted into the system, the requesting thread invokes QoS negotiation by calling *force_negotiation()*. As a result, the QoS levels of already-admitted tasks are recalculated and a new value for unfulfilled potential reward is computed. The overhead of the *force_negotiation()* call is charged to the caller.

In the current implementation, all created tasks execute in the same address space. The application is compiled into a single executable image that is loaded in its entirety at system start time. The code itself is thus static, although arrival/activation times at different nodes may vary dynamically.

5.2 Invocation Migration

On top of *qthreads*, we provide an invocation migration mechanism to implement the distributed QoS optimization protocol described in Section 4. The mechanism is completely transparent to the application. We call it *invocation* migration because the transfer occurs between two successive invocations of a periodic task (i.e., when one invocation has terminated and the next hasn't started yet). When the distributed QoS optimization heuristic determines that a task is to be migrated, the *state variables* of each thread in the transferred task are sent to the new machine and the

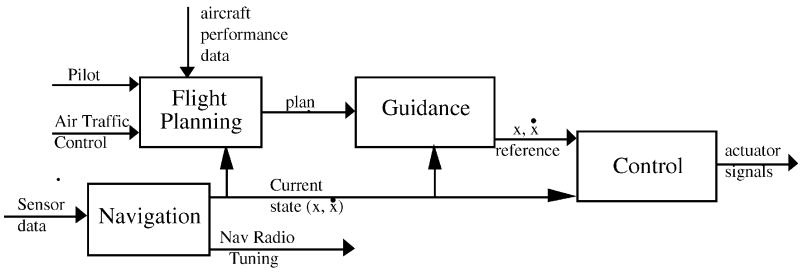

Fig. 5. Flight management system functions.

threads belonging to the task are destroyed at the source and recreated with the transferred state at the target. In the current implementation, state variables of a thread must be indicated to **RTPOOL** using a corresponding library call at thread initialization time. The *force_negotiation()* primitive is called on source and target after the transfer to update QoS levels accordingly. If a task must execute on a certain machine, the task can be *wired* to that machine by calling a *wire_task()* primitive.

5.3 Pool Membership API

A membership algorithm is used to maintain a consistent view of the current membership of the shared resource pool. Our group membership algorithm is a derivative of [51]. The user interface to that algorithm is the *subscribe_to_pool()* call which causes the machine on which the call is executed to join the named pool. When a new machine subscribes (joins), each machine in the pool adds the new member to the group. Since the new machine does not run any application task, its unfulfilled potential reward is zero. In our load-sharing heuristic, machines whose unfulfilled potential reward is above a given threshold will attempt to offload tasks to the new member. Task transfer will continue until the unfulfilled potential reward is balanced within a certain threshold, which stops the distributed QoS optimization protocol. When a machine crashes, the group leader (the machine with the highest number in the pool) recreates the destroyed tasks, then the load-sharing heuristic redistributes the load if necessary. When the group leader crashes, its successor (the machine with the next highest pool number) becomes the leader. Note that this mechanism is not an alternative to redundancy. Task state will be lost in case of a crash, but it can be avoided by task replication.

5.4 Communication API

An application need not be aware of where each of its tasks is executing. The same executable application image is started on every machine that joins the pool. The application is composed of tasks and the decision of where to run each task is left up to the load-sharing heuristic. This requires location-independent *send()* and *receive()* primitives for intertask communication. Tasks may communicate via local communication buffers if they are colocated on the same machine. Otherwise, an intertask message is sent across the network to the destination. Our communication protocol stack is implemented using xKernel 3.2 [52] and is layered on top of a UDP/IP stack. The communication subsystem architecture on each host is designed to support prioritized, bounded-time message delivery. This architecture has been proposed earlier in the context of implementing real-time channels [53]. We adapt it to export the abstraction of a sporadic communication server. The server is implemented as a separate task using *qthread* support. Currently, this task has only one QoS level. In the future, we will extend this architecture so that the communication QoS can also be negotiated.

6 Application—Aircraft Flight Control

We have used **RTPOOL** to provide negotiable timeliness guarantees for several real-time tasks required in our fully automated flight control system. This system was used to fly a simulated model of an F-16 fighter aircraft. Details of the automated aircraft flight problem are provided in Section 6.1, followed by a description of a method to determine the involved task QoS levels and rewards from application domain knowledge (Section 6.2). Section 6.3 summarizes the set of tasks, QoS levels, and rewards that describe the application.

6.1 The Automated Flight Control System

To familiarize the reader with our application domain, this section provides an introduction to automated flight systems, then highlights the particular control system we use during flight simulation experiments. Current Flight Management Systems (FMS) perform several flight control functions, including flight planning, navigation, guidance, and control [54]. Fig. 5 illustrates these FMS tasks and their interconnections; details of each module are provided in [54] and [55]. In such an FMS, real-time execution guarantees exist for the navigation, guidance, and control modules, allowing critical function deadlines to be met. Schedulability guarantees for these systems are typically computed off-line. Our QoS-negotiation scheme will allow the system to gracefully degrade performance when enough resources are lost to violate the off-line guarantees. In this paper, we consider the case where all tasks have a known bounded execution time. Issues in dealing with potentially unbounded on-line computations, such as run-time intelligent mission planning, are discussed in [56] and [57].

An aircraft flies in three-dimensional space, but travel within these dimensions is restricted because the aircraft is controlled using strictly aerodynamic forces and engine thrust. FMS aircraft guidance commands are typically issued in terms of aircraft altitude, airspeed, and compass heading. In our experiments, we control the aircraft using constant climb, cruise, and descent airspeeds, then employ

a simple "Guidance" function to alter commanded altitude and heading.

To achieve the altitude (z_{ref}) and heading (h_{ref}) specified by the "Guidance" function, we employ a control loop to compute *primary* actuator commands, including elevator, ailerons, rudder, and throttle. The elevator, ailerons, and rudder generate aerodynamic forces that directly affect aircraft roll and pitch attitude and, via dynamic coupling, alter aircraft heading and airspeed. The engine throttle provides a force along the aircraft fuselage which is used in combination with the aerodynamic forces to alter aircraft airspeed and altitude. Our controller is also capable of commanding a *secondary* set of actuators that improves flight performance, but is not critical for flight safety. Secondary actuators include the F-16's afterburner for extra engine thrust, as well as wing flaps and a speed brake used to enhance slow-airspeed control.

In a parallel research effort [2], a set of linear controllers have been implemented to calculate the *primary* actuator commands to achieve the desired reference altitude (z_{ref}) and heading (h_{ref}) for the aircraft. Controller state includes altitude (z), heading (h), pitch angle (p), and roll angle (r). Equation (6.1) shows the control laws used during our experiments, adopted from [2] and [56]. Because engine response time is slow, the throttle was not part of these control laws, but instead was preset based on "phase of flight" (e.g., throttle set to 100 percent for the departure climb, 75 percent for cruise, etc.). When executing at higher-performance QoS levels (see Section 6.3), the controller also exerts control over the set of *secondary* actuators using discrete-valued commands as described in [56].

$$\begin{pmatrix} elevator \\ ailerons \\ rudder \end{pmatrix} = \begin{pmatrix} K_1 & 0 & -K_{p_1} & -K_{d_1} & 0 & 0 \\ 0 & K_2 & 0 & 0 & -K_{p_2} & -K_{d_2} \\ 0 & K_3 & 0 & 0 & -K_{p_3} & -K_{d_3} \end{pmatrix} \begin{pmatrix} (z_{ref} - z) \\ (h_{ref} - h) \\ p \\ \dot{p} \\ r \\ \dot{r} \end{pmatrix}. \quad (6.1)$$

6.2 Computing QoS Levels and Rewards

Our QoS-negotiation scheme enables the application domain expert to express application-level semantics to **RTPOOL** using QoS levels, rewards, and rejection penalty. In this section, we briefly highlight how this support may complement mission planning techniques in the context of CIRCA, the Cooperative Intelligent Real-time Control Architecture ([1], [2]). Based on a user-specified domain knowledge base, CIRCA's main goal is to build a set of control plans to keep the system "safe" (i.e., avoid catastrophic failures such as an aircraft crash) while working to achieve its performance goals (e.g., arrive at its destination on time). In order to deal successfully with an inherently nondeterministic, perhaps poorly modeled, environment of a complex real-time system CIRCA employs *probabilistic* planning which models the system by a set of states and transition probabilities. System failure is modeled by temporal transitions to failure states (TTFs). CIRCA's mission planner uses its domain knowledge base to select appropriate actions (tasks) and their timing constraints (QoS levels) so that the probability of TTFs is reduced below a certain threshold. The reward decrease corresponding to degrading a task from one QoS level to another, or rejecting a task altogether, is computed from the corresponding increase in failure probability.

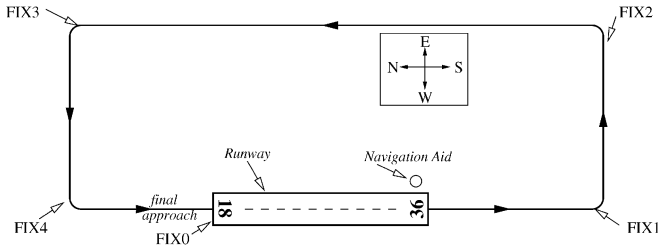

Fig. 6. Aircraft flight pattern flown during testing.

For example, the planner computes a maximum period for each task based on the notion of preempting TTFs [1]. For any state, an outgoing TTF is considered to be preempted if its probability is below the specified probability threshold, as described in [2]. To define alternative QoS levels, CIRCA's planner may compute different task periods based on a set of alternative TTF probability thresholds. For example, say a TTF has a cumulative probability distribution that reaches the threshold value when the preemptive task's maximum period is set to 0.2 seconds. But, suppose we need to relax the task's period requirement under overload. The new, longer period for degraded QoS is computed from the next higher probability threshold level and this task is assigned a lower reward that corresponds to the reduction in certainty that the TTF will be preempted. A complete set of task QoS levels may be developed by considering each TTF probability threshold.

6.3 Description of Flight Tasks

We have used the Aerial Combat (ACM) F-16 flight simulator [58] for all flight tests. ACM runs on a Sun workstation with a socket connection to the real-time execution platform. We have tested the QoS-negotiation capabilities by flying the simulated aircraft around the lefthand pattern illustrated in Fig. 6. In this pattern, the aircraft executes a takeoff and climb, then holds a constant altitude as it continues around a rectangular course through the descent and final approach to landing. By varying periods of the controllers and sensors, we are able to observe the degradation in flight quality (i.e., stability) as a function of each task's selected QoS level.

In this section, we describe the tasks and associated rewards used during our tests of the QoS negotiation algorithms. The goals of our example mission were to complete the flight around a rectangular pattern (illustrated in Fig. 6) and to destroy observed enemy targets, if any, using the simulated F-16's onboard radar and missiles. Four separate tasks were required to control the aircraft during flight: "Guidance," "Control," "Slow Navigation," and "Fast Navigation." These tasks function much like their similarly named FMS counterparts in Fig. 5. The

TABLE 1
Flight Plan with Different QoS Levels

Task	Level	Reward	Exec Time (ms)	Period (sec)	Module (Version)
Guidance	0	10	100	10	default
	1	15	100	5	default
	2	20	100	1	default
Controller	0	1	80	5	secondary
	1	100	60	1	primary only
	2	104	80	1	secondary
	3	120	60	0.2	primary only
	4	124	80	0.2	secondary
Slow Navigation	0	10	100	10	default
	1	20	100	5	default
	2	25	100	1	default
Fast Navigation	0	1	60	5	default
	1	100	60	1	default
	2	120	60	0.2	default
Missile Control	0	1	500	10	default
	1	30 (200)	500	1	default

"Guidance" task is responsible for setting the reference trajectory of the aircraft in terms of altitude and heading. The "Control" task is responsible for executing the closed-loop control functions that compute actuator commands, as described above in (6.1). We have two "Navigation" tasks that read sensor values, distinguished by the required update frequency. The navigation sensor values are used by the "Guidance" task to determine when and how to alter the commanded trajectory and are used as standard state feedback by the "Controller" task.

Table 1 shows the set of QoS levels present for all tasks, including the associated reward, execution time, period, and version. In our simple tests, we set each task deadline equal to its period, although there are no such requirements in our QoS negotiation protocol. Also, because each of these tasks is considered critical to execute (at least at a degraded QoS level), we set all task rejection penalties sufficiently high that all tasks are always accepted by the QoS negotiator.

In addition to the basic flight control tasks discussed above, we simulate a function necessary during military operation: "Missile Control." The "Missile Control" task is composed of two precedence-constrained threads: "Read Radar" and "Fire Missile." The "Read Radar" thread monitors aircraft radar to detect approaching enemy targets, then, if a target has been detected, the "Fire Missile" thread is used to launch a missile at the enemy target. As shown in Table 1, the simulated "Missile Control" task is computationally expensive and has two QoS levels. If Level 1 is possible, radar will be scanned with sufficient frequency to allow most any enemy target to be detected and destroyed. Otherwise (level 0), fast-moving targets may not be destroyed. During experiments (see Section 7.3), we varied the reward for "Missile Control" QoS Level 1 depending on the "subjective" relative importance of taking down enemy targets vs. flight control performance.

As described above, the "Controller" task is responsible for executing the control loop. At each invocation, the controller uses the (6.1) control law with appropriate gains to compute *primary* actuator outputs. Two versions of this function were tested, one that used the *secondary* actuators (QoS levels 0, 2, and 4) and one that did not (QoS levels 1 and 3). Use of these actuators allows the aircraft to perform better in terms of takeoff distance and climb rate, as shown in Section 7, at the expense of a longer task execution time. The importance of controller task period is illustrated by the relatively high reward given to the low-period QoS levels for the "Controller" task. The small reward changes between the use of the different versions (e.g., level 3 vs. level 4) reflects the fact that version choice is not critical for safety.[3]

The "Slow Navigation" task is responsible for reading sensors that do not require a high sampling rate. All navigation sensors are grouped into this task because they are used by the "Guidance" task to determine the high-level altitude and heading commands, but not by the more safety-critical "Controller" task. The Table 1 reward/period

3. We defined a QoS "level 0" for the "Controller" and "Fast Navigation" tasks that, as will be shown in Section 7, were so slow that the aircraft becomes unstable during turning maneuvers. Theses levels are included among their task's QoS negotiation options for illustrative purposes only and would not be there otherwise.

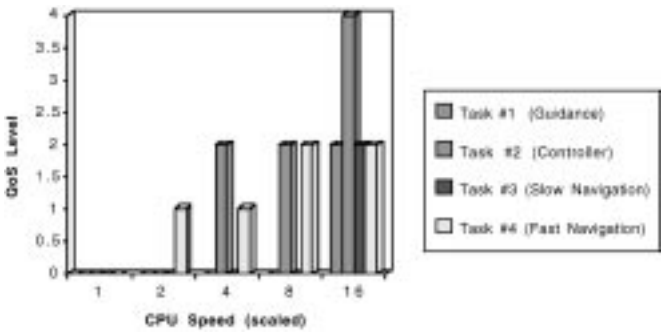

Fig. 7. QoS levels selected vs. CPU speed for flight tasks.

values for "Slow Navigation" reflect the noncritical nature of this task. Finally, the "Fast Navigation" task is responsible for updating all sensor data used by the "Controller" task. Since the system must read this data frequently to maintain sufficient state variable accuracy, the periods and rewards are similar to those used by the "Controller" task.

7 Evaluation

In this section, we show results illustrating how QoS negotiation can help aircraft flight control degrade gracefully. First, we assess the QoS negotiation heuristic for our set of flight tasks by observing how the QoS of each task degrades with lower machine speeds. In Section 7.2, we study aircraft performance during flight as a function of the "Controller" task's QoS level, illustrating graceful performance degradation by example. In Sections 7.1 and 7.2, we focus on tests that use a single machine and consider only the guidance, navigation, and control tasks. We conclude our experiments (Section 7.3) with tests which also include the missile control task and observe the effects of load sharing between two machines, with processor failure used to demonstrate graceful performance degradation.

7.1 QoS Negotiation Heuristic Testing

In Section 4, we described a simple local QoS optimization heuristic to help a service provider select a high-reward set of QoS levels for its clients. Using the QoS levels and rewards listed in Table 1, we illustrate the behavior of the presented heuristic. In this experiment, we kept the task set fixed and decreased the underlying CPU speed (increasing task execution times), then observed the corresponding decrease in task QoS levels. Fig. 7 plots the observed QoS levels versus CPU speed, normalized by the minimum CPU speed for which the task set is schedulable.

As shown in Fig. 7, Tasks 1 and 3 immediately degrade to QoS level 0 as soon as all "best" levels are no longer possible. This results primarily because Tasks 1 and 3 are less critical, so the penalty of their degradation is not as great. This effect illustrates both the major strength and weakness of the current QoS negotiation heuristic. As should be the case based on reward structure, Tasks 1 and 3 have their QoS levels reduced first because they are less critical. However, these tasks are degraded more than they should be in an optimal solution because the heuristic does not use any information about the semantics of QoS level parameters. For example, it does not "understand" the execution time and period of a task (and, thus, the task's computing requirements). Instead, it degrades QoS levels of clients based only on their rewards. So, it continues degrading tasks 1 and 3 until their minimum QoS level eventually reduces the QoS level of task 2, the primary time-consuming, low-period task, at which time the task set becomes schedulable.

Had the heuristic been able to "interpret" the QoS parameters of task 2, it would have been able to degrade it earlier. Not interpreting these parameters, however, allows complete separation between the schedulability analysis algorithm and the QoS optimization heuristic, as noted in Section 4. By using only reward information in its search for a feasible set of QoS levels, the same heuristic becomes applicable in any service that uses our QoS negotiation scheme. Only the schedulability analysis algorithm needs to change, in accordance with the semantics of QoS level parameters that define the service.

The purpose of the above example is to illustrate the compromise involved between the optimality of QoS negotiation and the convenience of minimizing dependencies between it and schedulability analysis. We also emphasize the separation between our QoS negotiation scheme as a general mechanism and any specific policies/heuristics used within its framework for a particular implementation.

7.2 Aircraft Performance

We evaluated the performance of our system by studying its ability to control the aircraft simulator during flight. In this section, we consider only the flight control tasks as they execute on one machine, saving discussion of the load sharing protocol and missile control task for the next section. As shown in Fig. 7, since the "Controller" and "Fast Navigation" tasks required the smallest execution period, these tasks are the bottlenecks for execution, so changes in aircraft performance are most easily observed by looking at changes in QoS levels for these tasks. Since these tasks are tightly coupled (i.e., the "Controller" task uses results from "Fast Navigation"), our test matrix included variations in the "Controller" task QoS level from its highest (4) to lowest (0) level and ensured that the "Fast Navigation" level acted with at least as low a period as was present in the "Controller" level.

As shown in Table 1, "Controller" task QoS levels are a function of two variables: task period and version. We present tests that illustrate major performance differences due to each of these variables, specifically during the critical takeoff/climb phase of flight. Fig. 8 illustrates differences between the two versions of the "Controller" task in their "best performance" case (period = 200 msec). Level 4 (with *secondary* actuation) requires a larger "Controller" task execution time than level 3 (no *secondary* actuation), thus it is harder to schedule. Climb performance with level 4 is only slightly better than that with level 3, consistent with their small reward difference. This example illustrates how QoS negotiation can achieve graceful degradation. Overall processor utilization is decreased by reducing the "Controller" task to level 3, but safety (i.e., controller stability) is not compromised.

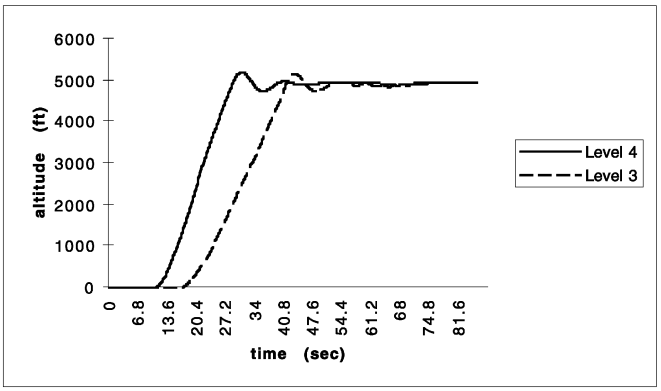

Fig. 8. Aircraft altitude performance with and without *secondary* control actuation.

Next, we performed tests with varying "Controller" task period. We isolated version from period effects by exclusively selecting QoS levels with *secondary* actuation (levels 0, 2, and 4), although similar trends result with the other task version. To illustrate performance changes as a function of task period, we consider three different QoS levels: level 4 with a period of 0.2 seconds (200 msec), level 2 with a period of 1 second, and level 0 with a period of 5 seconds. We include level 0 among the Controller's negotation options as a comparative example illustrating controller instability. Of course, no unstable QoS levels should be defined among a client's negotiation options since the client should not "ask" for instability.

Figs. 9, 10, 11, and 12 show state variables as a function of time from takeoff, climb, and a turn to East after reaching FIX 1 (see the pattern in Fig. 6). Fig. 9 shows the aircraft altitude for the different controller periods. As period increases, climb performance gracefully degrades between levels 4 and 2, but then becomes unstable in level 0 (period = 5 sec), illustrating the necessity of real-time response for the "Controller" task. Fig. 10 shows aircraft heading as a function of time for the three different "Controller" task periods during the same phases of flight. Again, heading control performance between "Controller" task levels 4 and 2 degrades, but remains stable, while level 0 results in an unstable response.

Figs. 11 and 12 show aircraft pitch angle and roll angle, respectively, for the two stable "Controller" QoS levels. Note that we do not include "Controller" level 0 here

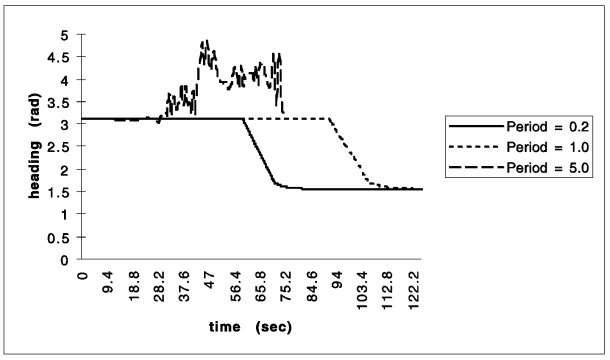

Fig. 10. Aircraft heading performance for different controller task levels.

because the instability obscures the other plots. Since pitch angle and altitude are coupled, the pitch angle has largest magnitude whenever the altitude is climbing (or descending) and, as illustrated in this plot, the increase in period to one second causes a large pitch angle to be required for a longer time, a stable, but undesirable, performance trait. Roll angle (Fig. 12) also shows a delay and longer roll angle deviation from zero for the slower-period control cycle, as well as significant overshoot when the task period increases.

7.3 Load Sharing—Flight with Missile Control

Load sharing capabilities are implemented in RTPOOL and we performed a final set of tests which included both the flight control tasks (with performance characteristics shown above) and a missile control task, as described in Section 6.3. In these tests, we start the system with two machines available for task execution. Because, as defined in Table 1, the missile control task was computationally expensive, the load sharing protocol places all flight control tasks on one machine and the missile control task (both the "Read Radar" and "Fire Missile" threads) on the other machine.

When the two machines function normally, both the flight and missile control tasks run in their maximum performance levels. In this case, enemy targets are quickly detected and fired upon, while flight control is identical to the best performance profiles in the Section 7.2 plots. For the next test set, we began operation with two functioning machines, then shut one down (simulating machine failure) just after takeoff. This requires the load sharing algorithm to function dynamically such that the one functional machine now has to execute both the flight and missile control tasks.

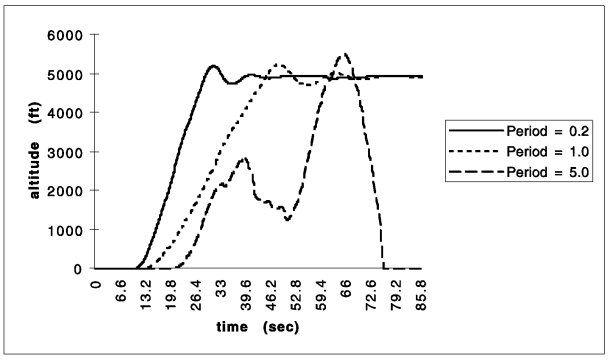

Fig. 9. Aircraft altitude performance for different controller task levels.

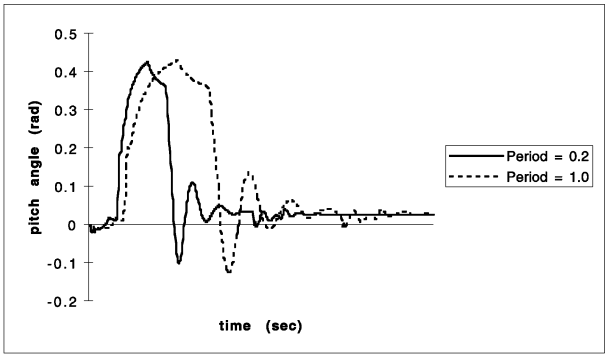

Fig. 11. Aircraft pitch performance for different controller task levels.

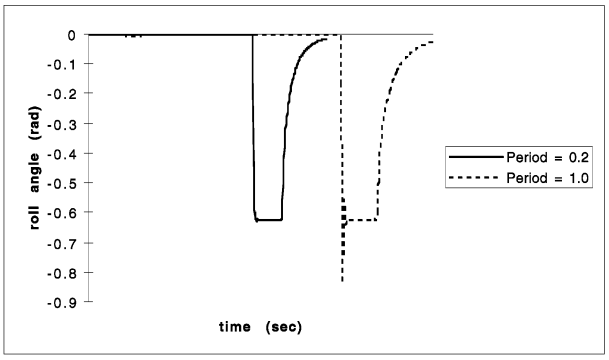

Fig. 12. Aircraft roll performance for different controller task levels.

To illustrate the importance of the relative rewards assigned to flight vs. missile control functions, we varied the missile control reward for QoS level 1 as shown in Table 1 and then ran the simulation for each of these two rewards. With the relatively low "Missile Control" reward, the system chooses to degrade the "Missile Control," "Guidance," and "Slow Navigation" functions to level 0, but manages to keep the "Controller" and "Fast Navigation" tasks safe levels (i.e., levels 2 and 1, respectively). In this manner, the flight control is a bit sluggish, but stable (as illustrated in Section 7.2). The aircraft is unable to launch missiles at most targets since it only scans its radar (in the "Missile Control" task) once every 10 seconds.

Alternatively, this system may be aboard an expendable drone whose most important function is to destroy a target or attack enemy aircraft. In this case, the reward set may be structured such that the missile control task takes precedence over accurately maintaining flight control.[4] To illustrate such changes in the task reward set, we altered the reward for QoS level 1 of the "Missile Control" task to 200 (as shown in Table 1). Now, when the second machine shuts down, the QoS negotiator reduces all flight control levels to 0 since the missile controller is perceived as the most important task. After one machine fails, the aircraft eventually becomes unstable, but it is still able to quickly detect and respond to enemy targets that appear on radar.

It is important to note that, had we used traditional algorithms for schedulability analysis which do not allow negotiated QoS degradation, the system would have failed to guarantee/accept the entire task set on the same processor, leading to complete mission failure. Our QoS negotiation scheme allows our system to continue after processor failure at a set of degraded task QoS levels which correspond to the relative importance placed on each task.

8 Summary and Future Work

In this paper, we presented a novel scheme for QoS negotiation in real-time applications. This scheme is applicable for the design of real-time service providers, extending the interface of such services in that 1) it adopts a modified notion of request guarantees that allows for defining QoS compromises and supports graceful QoS degradation and 2) it provides a generic means to express application-level semantics to control how application QoS is to be degraded under overload or failure conditions. Our QoS negotiation method improves the guarantee ratio over traditional admission control algorithms and increases the application-level perceived utility of the system.

The proposed QoS-negotiation architecture has been incorporated into **RTPOOL**, an example middleware service which implements a computing resource manager for a pool of processors. The synergy between components of the service and the QoS-negotiation support has been illustrated. **RTPOOL** is used for a flight control application to demonstrate the efficacy of QoS negotiation. We demonstrated that the application does have negotiable parameters/constraints and can thus benefit from the added flexibility of negotiation. We also outlined a method by which application task QoS levels and their respective rewards can be analytically derived from system failure probability. QoS-negotiation support, while guaranteeing maximum QoS levels during normal operation, is shown to provide graceful QoS degradation in case of resource loss.

We have demonstrated how an application can benefit from the proposed QoS-negotiation scheme, but we have not analyzed the performance of different QoS optimization policies nor the general scope of their applicability. We are currently studying alternative QoS-optimization methodologies and the scalability of our QoS-negotiation approach. We are also considering ways to implement negotiable fault tolerance QoS, perhaps as an extension to **RTPOOL**. Finally, we are considering the development of generic schemes for quantifying perceived utility to compute reward and penalty values. Possible approaches include adapting performability analysis and using economic models for computing utility/costs.

4. In our tests, when the missile control takes precedence over flight control during single machine operation, the aircraft becomes unstable. This is more extreme than one might want for an actual system since one can't launch missiles if the aircraft has crashed.

Acknowledgments

The authors wish to thank Farnam Jahanian, Ashish Mehra, Anees Shaikh, and Wu Chang for sharing their opinions and insights during the development of this paper. The work reported in this paper has been supported in part by the US National Science Foundation under Grant IRI-9209031 and by the US Office of Naval Research under Grant N00014-94-1-0229.

References

[1] D.J. Musliner, E.H. Durfee, and K.G. Shin, "World Modeling for the Dynamic Construction of Real-Time Control Plans," *AI J.*, pp. 83-127, 1995.

[2] E.M. Atkins, E.H. Durfee, and K.G. Shin, "Plan Development in Circa Using Local Probabilistic Models," *Uncertainty in Artificial Intelligence: Proc. 12th Conf.*, pp. 49-56, Aug. 1996.

[3] J. Xu and D.L. Parnas, "Scheduling Processes with Release Times, Deadlines, Precedence and Exclusion Relations," *IEEE Trans. Software Eng.*, vol. 16, no. 3, pp. 360-369, Mar. 1990.

[4] J. Xu, "Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence, and Exclusion Relations," *IEEE Trans. Software Eng.*, vol. 19, no. 2, pp. 139-154, Feb. 1993.

[5] T. Shepard and M. Gagne, "A Pre-Run-Time Scheduling Algorithm for Hard Real-Time Systems," *IEEE Trans. Software Eng.*, vol. 17, no. 7, pp. 669-677, July 1991.

[6] T.F. Abdelzaher and K.G. Shin, "Optimal Combined Task and Message Scheduling in Distributed Real-Time Systems," *Proc. Real-Time Systems Symp.*, vol. 16, Dec. 1995.
[7] M. Alfano, A. Di-Stefano, L. Lo-Bello, O. Mirabella, and J.H. Stewman, "An Expert System for Planning Real-Time Distributed Task Allocation," *Proc. Florida AI Research Symp.*, May 1996.
[8] P. Altenbernd, C. Ditze, P. Laplante, and W. Halang, "Allocation of Periodic Real-Time Tasks," *Proc. 20th IFAC/IFIP Workshop*, Nov. 1995.
[9] J.L. Lanet, "Task Allocation in a Hard Real-Time Distributed System," *Proc. Second Conf. Real-Time Systems*, pp. 244-252, Sept. 1995.
[10] T.C. Lueth and T. Laengle, "Task Description, Decomposition and Allocation in a Distributed Autonomous Multi-Agent Robot System," *Proc. Int'l Conf. Intelligent Robots and Systems*, pp. 1,516-1,523, Sept. 1994.
[11] C.M. Hopper and Y. Pan, "Task Allocation in Distributed Computer Systems through an AI Planner Solver," *Proc. IEEE 1995 Nat'l Aerospace and Electronics Conf.*, vol. 2, pp. 610-616, May 1995.
[12] B.R. Tsai and K.G. Shin, "Assignment of Task Modules in Hypercube Multicomputers with Component Failures for Communication Efficiency," *IEEE Trans. Computers*, vol. 43, no. 5, pp. 613-618, May 1994.
[13] K.G. Shin and C.J. Hou, "Evaluation of Load Sharing in Harts with Consideration of Its Communication Activities," *IEEE Trans. Parallel and Distributed Systems*, vol. 7, no. 7, pp. 724-739, July 1996.
[14] S.M. Yoo and H.Y. Youn, "An Efficient Task Allocation Scheme for Two Dimensional Mesh-Connected Systems," *Proc. 15th Int'l Conf. Distributed Computing Systems*, pp. 501-508, 1995.
[15] S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, pp. 671-680, 1983.
[16] K. Tindell, A. Burns, and A. Wellings, "Allocating Hard Real-Time Tasks: An NP-Hard Problem Made Easy," *J. Real-Time Systems*, vol. 4, no. 2, pp. 145-166, May 1992.
[17] E. Wells and C.C. Caroll, "An Augmented Approach to Task Allocation: Combining Simulated Annealing with List-Based Heuristics," *Proc. Euromicro Workshop*, pp. 508-515, 1993.
[18] J.E. Beck and D.P. Siewiorek, "Simulated Annealing Applied to Multicomputer Task Allocation and Processor Specification," *Proc. Eighth IEEE Symp. Parallel and Distributed Processing*, pp. 232-239, Oct. 1996.
[19] S.T. Cheng, S.I. Hwang, and A.K. Agrawala, "Schedulability Oriented Replication of Periodic Tasks in Distributed Real-Time Systems," *Proc. 15th Int'l Conf. Distributed Computing Systems*, 1995.
[20] T.-S. Tia and J.W.-S. Liu, "Assigning Real-Time Tasks and Resources to Distributed Systems," *Int'l J. Minim and Microcomputers*, vol. 17, no. 1, pp. 18-25, 1995.
[21] S.S. Wu and D. Sweeping, "Heuristic Algorithms for Task Assignment and Scheduling in a Processor Network," *Parallel Computing*, vol. 20, pp. 1-14, 1994.
[22] K. Ramamritham, "Allocation and Scheduling of Precedence-Related Periodic Tasks," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, no. 4, pp. 412-420, Apr. 1995.
[23] A. Banerjea, D. Ferrari, B. Mah, M. Moran, D. Verma, and H. Zhang, "The Tenet Real-Time Protocol Suite: Design, Implementation, and Experiences," *IEEE/ACM Trans. Networking*, vol. 4, no. 1, pp. 1-10, Feb. 1996.
[24] D.D. Kandlur, K.G. Shin, and D. Ferrari, "Real-Time Communication in Multi-Hop Networks," *IEEE Trans. Parallel and Distributed Systems*, vol. 5, no. 10, pp. 1,044-1,056, Oct. 1994.
[25] A. Cambell, G. Coulson, and D. Hutchison, "A Quality of Service Architecture," *ACM Computer Comm. Review*, Apr. 1994.
[26] C. Volg, L. Wolf, R. Herrwich, and H. Wittig, "HeiRAT—Quality of Service Management for Distibuted Multimedia Systems," *Multimedia Systems J.*, 1996.
[27] A. Lazar, S. Bhonsle, and K. Lim, "A Binding Architecture for Multimedia Networks," *J. Parallel and Distributed Computing*, vol. 30, pp. 204-216, Nov. 1995.
[28] K. Nahrstedt and J. Smith, "Design, Implementation, and Experiences with the OMEGA end-Point Architecture," *IEEE J. Selected Areas in Comm.*, Sept. 1996.
[29] K. Nahrstedt and J. Smith, "The QoS Broker," *IEEE Multimedia*, vol. 2, no. 1, pp. 53-67, 1995.
[30] C. Aurrecoechea, A. Cambell, and L. Hauw, "A Survey of QoS Architectures," *Proc. Fourth IFIP Int'l Conf. Quality of Service*, Mar. 1996.
[31] C. Mercer, S. Savage, and H. Tokuda, "Processor Capacity Reserves: Operating System Support for Multimedia Applications," *Proc. IEEE Int'l Conf. Multimedia Computing and Systems*, May 1994.
[32] C. Lee, R. Rajkumar, and C. Mercer, "Experiences with Processor Reservation and Dynamic QoS in Real-Time Mach," *Proc. Multimedia*, Mar. 1996.
[33] M. Jones, D. Rosu, and M.-C. Rosu, "CPUReservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities," *Proc. 16th ACM Symp. Operating Systems Principles*, Oct. 1997.
[34] C. Waldspurger, "Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management," PhD thesis, Massachusetts Inst. of Technology, Sept. 1995.
[35] P. Goyal, X. Guo, and H. Vin, "A Hierarchical CPU Scheduler for Multimedia Operating Systems," *Proc. Second Usenix Symp. Operating System Design and Implementation*, Oct. 1996.
[36] J.A. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Systems," *IEEE Software*, pp. 62-72, May 1991.
[37] S. Sommer and J. Potter, "Operating System Extensions for Dynamic Real-Time Applications," *Proc. IEEE Real-Time Systems Symp.*, pp. 45-50, Dec. 1996.
[38] M.B. Jones and P.J. Leach, "Modular Real-Time Resource Management in the Rialto Operating System," Technical Report MSR-TR-95-16, Microsoft Research, Advanced Technology Division, May 1995.
[39] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications," *IEEE J. Selected Areas in Comm.*, June 1997.
[40] R. Clark, E. Jensen, and F. Reynolds, "An Architectural Overview of the Alpha Real-Time Distributed Kernel," *Proc. USENIX Workshop Microkernels and Other Kernel Architectures*, 1992.
[41] H. Tokuda, T. Nakajima, and P. Rao, "Real-Time Mach: Towards a Predictable Real-Time System," *Proc. USENIX Mach Workshop*, pp. 73-82, Oct. 1990.
[42] G. Koren and D. Shasha, "D-Over: An Optimal On-Line Scheduling Algorithm for Overloaded Real-Time Systems," *Proc. IEEE Real-Time Systems Symp.*, pp. 290-299, Dec. 1992.
[43] S.K. Baruah, J. Haritsa, and N. Sharma, "On-Line Scheduling to Maximize Task Completions," *Proc. IEEE Real-Time Systems Symp.*, pp. 228-236, Dec. 1994.
[44] W. Zhao and K. Ramamritham, "Virtual Time CSMA Protocols for Hard Real-Time Communication," *IEEE Trans. Software Eng.*, vol. 13, no. 8, pp. 938-952, 1987.
[45] J.W.-S. Liu, W. Shih, K.-J. Lin, R. Bettati, and J. Chung, "Imprecise Computations," *IEEE Proc.*, Jan. 1994.
[46] P. Ramanathan, "Graceful Degradation in Real-Time Control Applications Using (m, k)-Firm Guarantee," *Proc. IEEE 27th Int'l Symp. Fault-Tolerant Computing*, June 1997.
[47] M. Bizzarri, P. Bizzarri, A. Bondavalli, F. Di-Giandomenico, F. Tarini, P. Laplante, and W. Halang, "Design of Flexible and Dependable Real-Time Applications," *Proc. 20th IFAC/IFIP Workshop*, Nov. 1995.
[48] R. Davis, S. Punnekkat, N. Audsley, and A. Burns, "Flexible Scheduling for Adaptable Real-Time Systems," *Proc. Real-Time Technology and Applications Symp.*, pp. 230-239, May 1995.
[49] C. Mercer, S. Savage, and H. Tokuda, "Processor Capacity Reserves: Operating System Support for Multimedia Applications," *Proc. IEEE Int'l Conf. Multimedia Computing and Systems*, pp. 90-99, May 1994.
[50] D. Seto, J.P. Lehoczky, L. Sha, and K.G. Shin, "On Task Schedulability in Real-Time Control Systems," *Proc. IEEE Real-Time Systems Symp.*, pp. 13-21, Dec. 1996.
[51] T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin, "RTCAST: Lightweight Multicast for Real-Time Process Groups," *Proc. IEEE Real-Time Technology and Applications Symp.*, June 1996.
[52] N.C. Hutchinson and L.L. Peterson, "The x-Kernel: An Architecture for Implementing Network Protocols," *IEEE Trans. Software Eng.*, vol. 17, no. 1, pp. 64-76, Jan. 1991.
[53] A. Mehra, A. Indiresan, and K.G. Shin, "Structuring Communication for Quality of Service Guarantees," *Proc. IEEE Real-Time Systems Symp.*, pp. 144-154, Dec. 1996.
[54] S. Liden, "The Evolution of Flight Management Systems," *Proc. 1994 IEEE/AIAA 13th Digital Avionics Systems Conf.*, pp. 157-169, 1995.

[55] J. Schreur, "B737 Flight Management Computer Flight Plan Trajectory Computation and Analysis," *Proc. Am. Control Conf.*, pp. 3,419-3,429, June 1995.
[56] E.M. Atkins, "Reasoning about and in Time when Building Plans for Safe, Fully-Automated Aircraft Flight," PhD thesis proposal, Dec. 1996.
[57] E.M. Atkins, E.H. Durfee, and K.G. Shin, "Buying Time for Resource-Bounded Planning," *Proc. AAAI-97 Workshop: Building Resource-Bounded Reasoning Systems Technical Report*, pp. 7-11, July 1997.
[58] R. Rainey, *ACM: The Aerial Combat Simulation for X11*, Feb. 1994.

Tarek F. Abdelzaher received his BSc (1990) and MSc (1994) degrees in electrical and computer engineering from Ain Shams University, Cairo, Egypt. He received his PhD (1999) degree from the University of Michigan in computer science with a specialization in real-time systems. He is an assistant professor of computer science at the University of Virginia. His current research interests include real-time systems, Quality of Service (QoS) control, networking, multimedia applications, next generation Web architecture, fault tolerance, and dependable computing. He is particularly interested in applying concepts from control theory and real-time computing to providing performance guarantees in open systems operating in complex unpredictable environments.

He is author or coauthor of more than 20 refereed publications. He received the Distinguished Achievement Award in Computer Science and Engineering from the University of Michigan in 1999. He is a co-editor of *IEEE Distributed Systems Online* and a guest editor of *Computer Communication* and the *Journal of Real-Time Systems*. He has also served on many conference committees and is the designated inventor of a European patent on Adaptive Web Servers. He is a member of the IEEE.

Ella M. Atkins earned SB (1988) and SM (1990) degrees in aeronautics and astronautics from the Massachusetts Institute of Technology, then worked as a structural dynamics test engineer at SDRC. She received her PhD degree (1999) from the University of Michigan in computer science and engineering with a specialization in intelligent systems. She is an assistant professor of aerospace engineering at the University of Maryland. At Michigan, she was a member of the Artificial Intelligence and Real-time Systems Laboratories and was software and computer systems lead for the Michigan Uninhabited Aerial Vehicle (UAV) project. Dr. Atkins' current research efforts are directed toward the application of real-time intelligent automation techniques to UAVs and space robotics. She is a member of the IEEE.

Kang G. Shin received the BS degree in electronics engineering from Seoul National University, Seoul, Korea, in 1970, and both the MS and PhD degrees in electrical engineering from Cornell University, Ithaca, New York, in 1976 and 1978, respectively. He is a professor and founding director of the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, Michigan. He has supervised the completion of 40 PhD theses and authored/coauthored more than 600 technical papers and numerous book chapters in the areas of distributed real-time computing and control, computer networking, fault-tolerant computing, and intelligent manufacturing. He has coauthored (jointly with C.M. Krishna) a textbook *Real-Time Systems* (McGraw-Hill, 1997). In 1987, he received the Outstanding IEEE Transactions on Automatic Control Paper Award, and Research Excellence Award in 1989, Outstanding Achievement Award in 1999, and Service Excellence Award in 2000 from The University of Michigan. In 1985, he founded the Real-Time Computing Laboratory, where he and his colleagues are investigating various issues related to real-time and fault-tolerant computing.

His current research focuses on Quality of Service (QoS) sensitive computing and networking with emphases on timeliness and dependability. He has also been applying the basic research results to telecommunication and multimedia systems, embedded systems, and manufacturing applications.

From 1978 to 1982, he was on the faculty of Rensselaer Polytechnic Institute, Troy, New York. He has held visiting positions at the US Airforce Flight Dynamics Laboratory, AT&T Bell Laboratories, Computer Science Division within the Department of Electrical Engineering and Computer Science at the University of California at Berkeley, and International Computer Science Institute, Berkeley, California, IBM T.J. Watson Research Center, and Software Engineering Institute at Carnegie Mellon University. He also chaired the Computer Science and Engineering Division, EECS Department, The University of Michigan for three years beginning in January 1991.

He is an IEEE fellow and member of the Korean Academy of Engineering, was the general chair of the 2000 IEEE Real-Time Technlogy and Applications Symposium, the program chairman of the 1986 IEEE Real-Time Systems Symposium (RTSS), the general chairman of the 1987 RTSS, the guest editor of the August 1987 special issue of *IEEE Transactions on Computers* on real-time systems, a program cochair for the 1992 International Conference on Parallel Processing, and served on numerous technical program committees. He also chaired the IEEE Technical Committee on Real-Time Systems during 1991-1993, was a distinguished visitor of the IEEE Computer Society, an editor of the *IEEE Transactions on Parallel and Distributed Systems*, and an area editor of the *International Journal of Time-Critical Computing Systems and Computer Networks*.